Global Concurrency Control in Heterogeneous Distributed Database Systems

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Weimin Du

Yungho Leu

Report Number:
90-1018


This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
GLOBAL CONCURRENCY CONTROL IN
HETEROGENEOUS DISTRIBUTED DATABASE SYSTEMS
Ahmed K. Elmagarmid
Weimin Du
Yungho Leu
CSD-TR-1018
September 1990
Global Concurrency Control in Heterogeneous Distributed Database Systems*

Ahmed K. Elmagarmid, Weimin Du and Yungho Leu
Indiana Center for Database Systems
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

We survey, in this paper, global concurrency control strategies for heterogeneous distributed database systems. We focus on the issues of consistency, local autonomy and performance. According to whether a strategy prevents or resolves inconsistency, it is classified into one of the two basic approaches: optimistic or pessimistic. The former intends to provide a high degree of concurrency among global transactions, while the latter is concerned with aborts of global transactions. The strengths and weaknesses of the two approaches are discussed.

1 Introduction

A heterogeneous distributed database system (HDDBS) is a federation of pre-existing database systems (called local database systems, or LDBSs). An HDDBS is the natural result of the shifting priorities and needs of an organization as it acquires new database systems that are designed independently. For many applications, an HDDBS is an attractive alternative to a single, integrated database system. An HDDBS is different from a set of database systems in that it supports global applications accessing multiple systems simultaneously. It is also different from traditional homogeneous distributed database systems in that it interconnects LDBSs in a bottom up fashion, thereby allowing existing applications developed on each of the LDBSs continue to be executable without modification.

An important feature of HDDBSs is autonomy of LDBSs. Local autonomy defines the right of each LDBS to control access to its data by other LDBSs and the right to access and administer its own data independently of other LDBSs. As a result, LDBSs may use different data models, different concurrency control strategies and they can schedule accesses to its data independently. Local autonomy is desirable and necessary in HDDBSs to guarantee that old applications are executable after interconnection, to facilitate flexible interconnection of various kinds of LDBSs, and to ensure the consistency and the security of LDBSs.

*This work is supported by a PYI Award from NSF under grant IRI-8857952, a David Ross Fellowship from Purdue Research Foundation, and grants from AT&T Foundation, Mobil Oil, SERC and Tektronix.
Concurrency control in HDDBSs is much different from that in homogeneous distributed database systems, due to existence of local concurrency controllers (LCCs). An LCC resides at each LDBS and maintains its consistency. They, however, are not capable of maintaining the consistency of the global database, because global transactions may be scheduled inconsistently at different sites. In order to prevent this kind of inconsistency, a global concurrency controller (GCC) is needed. The GCC is built on top of LCCs coordinating local executions at different sites.

Concurrency control in HDDBSs is also more difficult than that in homogeneous distributed database systems, due to the autonomy of LCCs [DEK90]. The LCCs are independently designed and cannot be modified because of the autonomy restrictions placed by the LDBSs. In addition, the LCCs have the right to schedule local and global transactions independently, based on their own considerations. The only control the GCC has over local executions is submissions of global transactions. However, it is possible that a global transaction effectively precedes another even if it is executed entirely after the latter at all local sites (see [BHG87]).

The necessity and difficulties of the global concurrency control in HDDBSs were recognized by Gligor, et al. as far back as 1984 [GL84]. Several conditions for global concurrency control were identified in [GPZ86]. Since then, a large amount of work has been done in developing algorithms for global concurrency control. In this paper, we survey some of these algorithms and discuss their strengths and weaknesses in preserving local autonomy, maintaining the global consistency, and performance they provide.

The remaining sections are organized as follows. We first discuss, in Section 2, three basic issues of global concurrency control in HDDBSs, namely, autonomy, consistency and performance. In Sections 3 and 4, we survey the various solutions for global concurrency control. Section 5 concludes the paper with a summary and suggestions for future research.

2 Consistency, Autonomy and Performance

The main motivation for concurrency control is to maintain database consistency and provide high performance by allowing concurrent execution of transactions. In HDDBSs, the global concurrency control mechanisms should also preserve the autonomy of LDBSs. In this section, we discuss the three issues (i.e., local autonomy, consistency and performance) in the context of HDDBS. We first give an HDDBS model. A classification of global concurrency control strategies is also presented in this section.

2.1 HDDBS Model

An HDDBS is a distributed database system consisting of LDBSs. Each local database is a set of data items. There are two kinds of transactions in an HDDBS. A local transaction accesses data items of one local database, while a global transaction accesses data items of more than one local database. A global transaction consists of a set of global subtransactions which access a single local database and are executed at the sites along with local transactions. Both local and global
transactions are correct in the sense that, when executing alone, they transform the corresponding database from a consistent state to another consistent state. More specifically, a global transaction transforms the global database from a consistent state to another consistent state, while a local transaction, as well as a global subtransaction transforms a local database from a consistent state to another consistent state.

The transaction processing model for HDDBSs is shown in Figure 1. It consists of a Global Data Manager (GDM), a Global Transaction Manager (GTM), a collection of LDBSs, a set of global and local transactions and a set of server processes. A server process runs at each site and represents the interface between the GTM and the LDBSs.

When a global transaction is submitted to the HDDBS, it is first decomposed and translated by the GDM into a set of subtransactions that run at the local sites where the referenced data reside. It is assumed that, for every global transaction, there is at most one subtransaction per site [GL81]. These subtransactions are then directed to the GTM. The GTM submits the subtransactions to the corresponding LDBSs and coordinates their executions at local sites so that the global database consistency is maintained. The GCC is a module of the GTM which is responsible for the global concurrency control.

2.2 Local Autonomy

Local autonomy, as we mentioned before, is an important feature of HDDBSs. It defines the ability of each LDBS to perform various kinds of operations on and to exercise various kinds of control over its own database. For example, an LDBS should be able to implement its own data model,
catalog management strategy, and its own naming convention. It should also be able to exercise control over local transactions and global subtransactions (e.g., to delay or abort a transaction) to maintain the consistency of the local database. Local autonomy also defines the right of each LDBS to make decisions regarding the service it provides to other LDBSs. Local autonomy is required to guarantee that local users can continue to run local applications on their LDBS regardless of interconnection and to ensure that the basic consistency, security and performance requirements of an LDBS are met while allowing other LDBSs to access its data.

Local autonomy is difficult to quantify. To discuss how well a global concurrency control strategy preserves local autonomy, we distinguish among different aspects of local autonomy. The ability of global concurrency control strategies to preserve these aspects of autonomy is discussed in the next two sections.

From transaction management point of view, the following three types of autonomy are identified [VPZ86]. Design autonomy reflects the fact that LDBSs were independently designed; execution autonomy refers to the ability of an LDBS to decide whether and how to execute transactions; and communication autonomy refers to the ability of an LDBS to decide whether and how to communicate with some or all other LDBSs.

Another way to study the effects of local autonomy on HDDBSs is to distinguish static and dynamic aspects of local autonomy. Static autonomy defines the right of an LDBS to make decisions regarding the static issues which are known at interconnection time. Static autonomy reflects the fact that LDBSs were designed and implemented independently. For example, the static aspect of design autonomy defines the right of an LDBS to choose its own data model and concurrency control strategy and manifests itself as heterogeneity in the LDBSs. The static aspect of execution autonomy, on the other hand, says that an LDBS has the right to specify the general execution policy (e.g. accessibility of global transactions to its data and priority at which they access data). That a GCC preserves the static aspect of local autonomy means that it imposes no restriction on LCCs (i.e., LDBSs may use any concurrency control strategies) and requires no modification to the LCCs. Dynamic autonomy, on the other hand, defines the right of each LDBS to make runtime decisions regarding the dynamic issues which are known or made at run time. For example, the dynamic aspect of execution autonomy defines the right of an LDBS to decide, based on the general policy and the current execution environment, whether and when to start, abort or commit a specific transaction. That a GCC preserves dynamic autonomy implies that it exercises no control over local executions except the submission of global subtransactions.

2.3 Consistency

As in homogeneous distributed database systems, the goal of global concurrency control in HDDBSs is to maintain the HDDBS consistency. Due to the hierarchical structure of HDDBSs, two kinds of consistency coexist: one that previously existed in each of the LDBSs (called local consistency) and the other that results from the interconnection process (called global consistency). Local consistency defines constraints on relationships of data in a local database and on interactions among local transactions and global subtransactions executed at the site, while global consistency
defines constraints on relationship of data at different local databases and on interactions among global transactions, as well as interactions among local transactions executed at different sites. We assume that the LCC at each site maintains the local consistency of the LDBS. More specifically, it guarantees the serializability of the local execution. It is the GCC’s responsibility to coordinate local executions to ensure that the global consistency requirements are met.

The conventional way of maintaining the global consistency is to execute global and local transactions in a serializable fashion. An execution of a set of transactions is serializable if it is equivalent to a serial execution of the transactions. Serializability in HDDBSs represents the strongest type of consistency in that it treats an HDDBS as a strongly coupled homogeneous distributed database system. There is no distinction between local and global transactions, nor between local and global consistencies. The problem with serializability in HDDBSs is that it is difficult to maintain, due to local autonomy of LDBSs [DELO89] [DEK90]. Since LCCs have the right to schedule local transactions and global subtransactions independently, the only control the GCC has over local executions is the submission of global transactions. As we have mentioned before, it is generally impossible to maintain the global serializability by just controlling the submission of global transactions.

It is also possible to maintain the global consistency using weaker correctness criteria, e.g., quasi serializability [DE89], in some applications. A global execution of a set of local and global transactions is quasi serializable if local executions are all serializable and it is equivalent to a quasi serial execution in which the global transactions are executed sequentially. Quasi serializability focuses on the interleavings of global transactions which the GCC is able to control and therefore is maintainable at the global level without violating local autonomy. Quasi serializability represents a type of HDDBS consistency weaker than that represented by serializability. More specifically, it guarantees that the execution of each global transaction appear to other global transactions as a single indivisible step. It is, however, possible that a local transaction does not appear indivisible to local transactions at other sites. Therefore, quasi serializability is most suited to those HDDBS applications where local transactions at different sites are independent (in other words, the execution of a local transaction does not affect that of other local transactions at another site) or interact in a partial order (see [ED90a]).

Global concurrency control strategies based on both serializability and quasi serializability are discussed in this paper.

2.4 Performance

Degree of concurrency is the main measure of performance for concurrency control in database systems. The degree of concurrency provided by a concurrency control protocol is measured by the possible interleavings of transactions it allows. In HDDBSs, we consider two kinds of interleavings: that among global transactions and that among local transactions and global subtransactions at a site. The latter is determined by LCCs. It is this goal of the GCC to provide a high degree of concurrency among global transactions. One way of achieving the goal is to impose as few restrictions as possible on submission and execution of global transactions.

Another important issue affecting performance is abort ratio of global transactions. In HDDBSs,
global transactions are aborted not only because of the conflicts in a local execution, but also because of the inconsistency of serialization orders between local executions. Consider two global transactions that both access two local databases. Suppose that they are submitted at about the same time. Due to the different scheduling policies two LDBSs use, it is possible that the two global transactions are scheduled in different orders at two sites. Generally, the more common sites two global transactions access, the more likely that they will be scheduled inconsistently. Such an inconsistency usually results in aborts of at least one global transaction.

Another reason that abort ratio of global transactions is important in HDDBSs is that the execution of a global transaction is usually expensive and sometimes undoable. The abort of a global transaction implies that all its subtransactions must be aborted. This, however, is very difficult (if not impossible) at some sites due to local autonomy.

Therefore, it is very important for a concurrency control strategy to abort as few global transactions as possible. This can be done in two ways. First, it should try not to abort global transactions because of local conflicts. In other words, the abort of a subtransaction should not result in the abort of the global transaction. Second, it should reduce the possibility of inconsistency between local executions, and in case an inconsistency occurs, aborts no more global transactions than necessary.

The trade-off between providing high degrees of concurrency and minimizing the number of global transactions aborted is an important decision in designing a global concurrency control algorithm, and results in two different approaches of global concurrency control.

2.5 Classification of Strategies

A global concurrency control strategy in HDDBSs can be classified as either pessimistic or optimistic. The optimistic approaches try to provide a high degree of concurrency, while the pessimistic approaches are concerned with abort ratio of global transactions.

Generally, pessimistic approaches attempt to prevent inconsistency by imposing restrictions on the submission of global transactions. The GCC assumes that there are conflicts among global transactions whenever such conflicts could possibly exist. The strategies in this group differ in the restrictions imposed on the submission of global transactions to guarantee a specific (serialization or quasi serialization) order. GCCs based on this approach do not abort global transactions due to inconsistency between executions. On the other hand, they allow low degrees of concurrency. In addition, they usually are unable to make use of dynamic information about local executions because it is impossible to predict in advance whether the information is available.

In contrast to pessimistic approaches, optimistic approaches do not control the submission of global transactions. Instead, they detect and resolve inconsistency after the execution of global transactions. The approaches are based on the assumptions that not every pair of global operations could conflict arbitrarily. The strategies in this group differ in how they detect and resolve inconsistency.
3 Pessimistic Approaches

In this section, we survey four pessimistic approaches of global concurrency: altruistic locking, top-down, site graph and access graph. One property they all share is that no global transaction is aborted due to inconsistency among local executions. Therefore, the main measure of performance is the concurrency degree of global transactions.

3.1 Altruistic Locking

The altruistic locking strategy was initially proposed for coping with long-lived transactions and is also used as a concurrency control strategy in HDDBSs [AGMS87]. The basic idea of the strategy is to prevent inconsistency by not allowing two global transactions to access a local database concurrently. A global transaction must lock a site before it can access the site. Once its all requests have been processed, it can release the lock on the site. Releasing a lock is a conditional unlocking operation. Other global transactions waiting to lock the released site may be able to do so if they are willing to abide by the following rules. (1) No two global transactions hold locks on the same site simultaneously unless one of the transactions locked and released the site before the other locked it. In other words, the later lock-holder is in the wake of the releasing transaction. (2) If a transaction is in the wake of another transaction, it must be completely in the wake of the transaction. In other words, if G1 locks site A which has been released by G2, then anything currently locked by G1 must have been released by G2 before it was locked by G1. (This requirement is relaxed once G2 finishes.) Note that the releasing operation is not two-phase, i.e. the global transaction is free to continue to lock new sites after it has released locks; while the unlocking operation is two-phase and usually is done after global transactions complete.

Example 1. Consider an HDDBS consisting of five LDBSs: A, B, C, D, and E. Let G1 be a global transaction which accesses sites A, C, D, and E in that order, G2 accesses sites A and C, and G3 accesses sites A and B. Suppose that at current state, G1 hold the locks on sites A, C and D. The sites A, C and D constitute the current wake of G1. After G1 released the locks, G2 is allowed to access sites A and C, since they are in the wake of G1. G3, however, can only access site A, and has to wait until G1 completes and unlocks all its locks before it can access site B, which is outside the wake of G1.

The execution order of global transactions is decided in the following way. If a global transaction is in the wake of other global transactions, then its order is after those global transactions. For global transactions which are not in the wake of each other, their order is decided by the order they lock the conflict sites. It is shown that the altruistic locking maintains the same execution order among global transactions at all sites.

The main advantage of this strategy is that it exercises no control over and requires no information about local executions. In other words, it preserves local autonomy of LDBSs. However, the strategy provides no concurrency among global transactions because they are executed sequentially at each site. The benefit of doing this is that the execution order of global transactions at local sites are always consistent. Another problem is that it maintains global serializability only.
if all LCCs maintain strict serializability of local executions. In other words, serialization order of global subtransactions is compatible with their execution order. This is however not true for some concurrency control protocols (e.g., serialization graph testing). On the other hand, it has been shown in [DE89] that it maintains quasi serializability regardless of concurrency control protocols LCCs use.

3.2 Top-down

As mentioned in [GPZ86], one way of performing global concurrency control in HDDBSs is the top-down approach. In top-down approach, the GCC determines the execution order of global transactions before their submission to local sites which is then enforced at each site. There are two basic steps in a top-down approach of concurrency control: (1) determining an order at global level, and (2) enforcing the order at local level.

The non-two-phase locking protocol proposed by Vidyasankar in [Vid87] gives a solution to the first step. It provides a way of dynamically determining the order as global transactions are executed. This approach is adapted from the well known non-two-phase locking protocol for the traditional concurrency control problem [SK80].

The protocol applies to those HDDBSs that are formed as a rooted tree. The subtransaction of a global transaction is treated as an atomic transaction step, and each LDBS is treated as a distinct data item. The GCC issues the subtransactions of a global transaction to the individual LDBSs according to the tree protocol. The following is a simple example illustrating the idea.

Example 2. Consider an HDDBS which is interconnected as a rooted tree in Figure 2. Let G be a global transaction and Gₐ be its subtransaction executed at site X. Suppose that subtransactions of G are to be executed at sites B, E, F and H. It first “locks” B and sends subtransaction Gₐ to site B. Then it locks E and F. After that, and after the execution of Gₐ completes, it may release the lock on B. Likewise, on completion of the execution of Gₐ and after locking H, it can release the lock on E. While the locks are acquired in the tree order, it can be released in any order.

In [ED90b], various techniques of enforcing the global execution order at local sites are discussed.
One way of doing this is to control the submission order of global subtransactions. The task is performed by server processes at local sites (called stub processes in [ED90b] and site queue in [LE90]). The server process guarantees that the LCC schedules global subtransactions in the order they are submitted. More specifically, the server process submits a global transaction to the LCC only if the serialization orders of all previously submitted global transactions have been determined by the LCC. The technique works for those LCCs that determine the serialization order of a transaction according to an event which can be identified before the termination of the transaction. An LCC with this property is said to be static [LE90]. Both two phase locking and time stamp ordering protocols are static. For example, in two-phase locking protocols, the serialization order of a transaction is determined once it reaches its lock point.

Unfortunately, not all concurrency control protocols are static, e.g., serialization graph testing. It is generally impossible to guarantee a specific serialization order using the above technique. On the other hand, the technique can always be used to enforce a quasi serialization order. The concurrency degree provided by the algorithm depends on local concurrency control strategies. If, for example, two-phase locking protocol is used, global subtransactions at the site are executed almost sequentially. A higher degree of concurrency is possible if timestamp ordering protocol is used. The price is, however, possible conflicts between global subtransactions, and therefore aborts of some subtransactions.

### 3.3 Site Graph

Site graph algorithm proposed in [BS88] is an attempt to maintain global serializability without imposing any restriction on LCCs. In this algorithm, the GCC maintains an undirected graph, called site graph, in which the nodes are sites and the edges are global transactions. When a global transaction $T$ is received, the GCC first determines the sites that contain copies of data accessed by the global transaction and connects them to form a linear link in the graph. The following example illustrates the notion of site graphs:

**Example 3.** Consider an HDDBS that contains data item $x$ at sites 1 and 2, $y$ at sites 1 and 3, and $z$ at sites 2 and 3. Let $G_1$ and $G_2$ be two global transactions:

$G_1 : T_1(x_1)w_1(y_1)w_1(y_3) \quad G_2 : T_2(y_3)w_2(z_2)w_2(z_3)$

Since the data are replicated, the GCC may generate one of the following sequences of local operations for each global transaction:

$G_1 : r_1(x_1)w_1(y_1)w_1(y_3) \quad G_2 : r_2(y_3)w_2(z_2)w_2(z_3)$

The site graphs for the sequences are shown in Figure 3.(a) and 3.(b).

In Figure 3, site graph 3.(a) is acyclic, while site graph 3.(b) contains a cycle. That a site graph is cyclic means that the execution of the global and local transactions may be non-serializable. For instance, the following non-serializable execution may occur if the second sequence is submitted for execution:

Site 1: $r_1(x_1)w_1(y_1)r_2(y_1)$
Site 2: \( w_2(z_2) \)
Site 3: \( r_1(y_3) w_1(z_3) w_2(z_3) w_1(z_3) \)

In this execution, local transaction \( L \) introduces an indirect order between \( G_1 \) and \( G_2 \) at site 1, which is different from their order of conflicting operations at site 1. While for the first sequence, the local transaction can not introduce indirect orders and therefore is guaranteed to result in a serializable execution.

The algorithm is the only pessimistic strategy that both maintains global serializability and does not violate local autonomy. However, it allows a lower degree of concurrency among global transactions. The reason is that the edges in a site graph cannot be purged even after the completion of the corresponding transaction, as illustrated by the following execution.

\[
E_1 : r_1(a) w_{g_1}(a) w_{g_2}(b) w_{g_1}(b) r_2(c) w_{g_2}(c) w_{g_3}(d) w_{g_2}(d) ... r_n(y) w_{g_n}(y) w_{g_{n+1}}(z) w_n(z)
\]

In this execution, \( G_1 \) executes entirely before \( G_2 \), which executes entirely before \( G_3 \) and so on. The serialization order, however, is the reverse: \( G_{n+1} \rightarrow G_n \rightarrow ... \rightarrow G_2 \rightarrow G_1 \). In other words, a global transaction (e.g., \( G_{n+1} \)) may effectively precedes another global transaction (e.g., \( G_1 \)) which executed arbitrarily long before.

### 3.4 Access Graph

The basic idea of the access graph algorithm proposed in [DE90] is similar to that of the site graph algorithm. The algorithm provides a higher degree of concurrency than the site graph algorithm. The price for this is, however, a lower degree of consistency. More specifically, it only guarantees quasi serializability in general.

The algorithm is based on the following property of quasi serializable executions. The quasi serialization order of two global transactions is compatible with their execution order if they do not interleave with each other. We say that two transactions interleave with each other if either they are executed concurrently (direct interleaving) or a third transaction directly interleaves with both of them (indirect interleaving). The algorithm ensures the quasi serializability of a global execution by maintaining the acyclicity of the access graph of the execution. An access graph characterizes the current execution environment from a single global transaction's point of view. More specifically, the access graph of an execution with respect to a global transaction is a subgraph of its site graph. The subgraph consists of only those edges that are introduced by global transactions who interleave (either directly or indirectly) with the global transaction. It has been shown that the quasi serializability of an execution is assured if its access graphs are acyclic with respect to all
global transactions.

The algorithm works by grouping global transactions. The transactions in each group form an acyclic access graph and therefore can be submitted to local sites without any control. Transactions at different groups, however, must be executed separately.

Example 4. Consider an HDDBS consisting of five LDBSs: A, B, C, D, and E. Let $G_1$, $G_2$ and $G_3$ be three global transactions where $G_1$ accesses A, B and C, $G_2$ accesses A and C, and $G_3$ accesses B, D and E. Suppose that they are submitted to the GCC in the order: $G_1$, $G_2$ and $G_3$. The GCC directs $G_1$ to the local sites immediately. $G_2$, however, cannot be submitted until $G_1$ finished because they form a cyclic access graph. Since $G_1$ and $G_3$ do not form a cyclic access graph, $G_3$ is submitted to local sites and executed concurrently with $G_1$ at site B. After both $G_1$ and $G_3$ have finished, $G_2$ will be submitted.

The main advantage of using quasi serializability as the correctness criterion is that the GCC only needs to consider those global transactions that are in the same group. Therefore, edges in an access graph are purged after all global transactions in the corresponding group have completed. As a result, it provides a higher degree of concurrency than the other three algorithms. In addition, it does not violate local autonomy. On the other hand, it is only suited to those HDDBS applications where quasi serializability is appropriate as a correctness criterion.

3.5 Discussion

The basic idea of pessimistic approaches is to prevent inconsistency between local executions from occurring. There are two ways of doing this. The first is to control the order at which global subtransactions are submitted, and the second is to control possible interactions among global transactions. The altruistic locking and top-down algorithms follow the first approach, while the site graph algorithm follows the second. The access graph algorithm combines the two approaches in the sense that it controls interactions among global transactions in a group and controls submission order of global transactions in different groups.

The main advantage of the first approach is the relatively high degree of concurrency (comparing to the second one), due to making use of static information of LCGs (i.e., when and how an LCC determines the serialization order of a transaction). For example, the altruistic locking algorithm allows a global transaction to access a site which is still locked by other transactions as long as they request no more work at the site. Similarly, the top-down algorithm allows a global transaction to access a site after the previous transactions' serialization orders have been determined (e.g., they reached their lock points). So a basic assumption behind the approach is that LCCs serialize global subtransactions in an order compatible with their submission order. This, generally, implies violation of static autonomy. Therefore, they are suited to only the HDDBS applications where the assumption is true.

The second approach (controlling possible interactions) makes no assumption on LCCs in maintaining global serializability. Therefore, it can be applied to all HDDBS applications. As we mentioned before, the approach suffers from low degree of concurrency.
The problems with both approaches may be solved if global serializability is not required. For example, if quasi serializability is used as the correctness criterion, the assumption of compatibility of submission order and serialization order is not necessary in the first approach, and the concurrency degree in the second approach can also be greatly improved (e.g., access graph algorithm).

4 Optimistic Approaches

The three algorithms discussed in this section share a common property of imposing no restriction on submission of global transactions. Theoretically, they can provide a very high degree of concurrency among global transactions. The price for this is possible aborts of global transactions. Due to the difficulties of detecting inconsistency, they may abort more global transactions than necessary. All three approaches use serializability as the criterion.

4.1 Decentralized Global Concurrency Control

Breitbart et al. proposed a decentralized global concurrency control algorithm for HDDBSs where each LDDS uses two-phase locking strategy [BLS89]. The basic idea of the algorithm is to maintain global serializability by synchronizing release of locks held by subtransactions of a global transaction. A global subtransaction will not release locks until it receives an “end-of-transaction” message issued by the GCC. The message is issued only if the global transaction finishes its execution at all local sites. Semantically, the “end-of-transaction” is equivalent to the “commit” message in two phase commit protocol. It has been shown that global serializability is assured if all global transactions follow this protocol.

On the other hand, global deadlocks may occur, due to inconsistency among local executions. Consider two conflicting global transactions \( G_1 \) and \( G_2 \) that both access sites 1 and 2. Suppose that \( G_1 \) is scheduled before \( G_2 \) at Site 1, but after \( G_2 \) at Site 2. Then \( G_2 \) waits for locks held by \( G_1 \) at Site 1, which in turn waits for locks held by \( G_2 \) at Site 2. The algorithm is optimistic in the sense that it detects inconsistencies (global deadlocks in this case) after the execution. Two algorithms have been proposed for deadlock detection. The first is a time-out based mechanism which checks the acyclicity of the potential conflict graph (PCG) of an execution. A PCG is a directed graph \( G = (V, E) \), where \( V \) consists of a set of global transactions and \( E \) consists of edges \( G_i \rightarrow G_j \) such that \( G_i \) is in the waiting state and \( G_j \) in the active state at a local site. The acyclicity of PCGs guarantees there is no global deadlock. The cyclicity of PCGs, on the other hand, may or may not be resulted from deadlocks. The second algorithm is based on the concept of value date [LT88]. The GCC associates with each global transaction a value date (timestamp) which denotes when the transaction is supposed to finish. Global transactions that pass their value date are aborted by the GCC, regardless of whether they are still running. Global deadlocks, therefore, are always broken after some of global transactions reach their value dates.

The algorithm makes a practical assumption in the sense that it requires all LDBSs to use two-phase locking. This, from a strict sense, violates design autonomy of LDBSs. No LDBS is allowed to join the HDDBS unless it uses two-phase locking as the concurrency control strategy.
In addition, both deadlock detection algorithms may detect global deadlocks that do not exist, resulting aborts of more global transactions than necessary. On the other hand, it maintains global serializability and provides a high degree of concurrency.

4.2 SuperDatabases

Pu proposed an optimistic approach called the SuperDatabases [Pu88]. The basic idea behind this approach is as follows. Each LCC executes the subtransactions as an ordinary local transaction. When a subtransaction is completed, the corresponding LCC reports to the GCC the serialization order called O-element. The GCC then constructs for each global transaction an order-vector (O-vector) as the concatenation of all O-elements of the subtransactions of the global transaction. The order on the O-vectors is defined in a strict sense. 0-vector(T_1) → 0-vector(T_2) if and only if for all LDBSs, O-element(T_1) → O-element(T_2). If a global transaction does not have a subtransaction at a particular site, a wild-card O-element, denoted by * (star) is used for the corresponding component of the O-vector. The order of this component does not matter and, by definition, O-element(any) → *, and *, → O-element(any). A certification is done for a global transaction when it tries to commit. It is done by comparing the O-vector of the committing global transaction against the O-vectors of the recently committed global transactions. If the new O-vector can find a place in the total order of the recently committed global transactions' O-vectors, it is committed; otherwise, it is aborted.

It is not hard to see that the algorithm maintains global serializability. It also provides a high degree of concurrency because it only aborts those global transactions that introduce inconsistency. In addition, it preserves static autonomy in the sense that it imposes no restriction on local concurrency control strategies LDBSs use.

However, two minor issues must be addressed before this algorithm can be widely used. The first problem is that the LCC may not be able to determine the serialization order of a transaction even after the completion of the transaction (see execution E1 in Section 3.3). Generally, the serialization order of a transaction depends not only on the execution of previously executed transactions but also on the execution of those executed later. The second problem is that it is not clear how the GCC can get O-elements of global subtransactions from LDBSs. Asking LCCs to report the order may imply, in some environments, violation of dynamic autonomy.

4.3 Optimistic Ticket

The optimistic ticket approach proposed by Dimitrios and Marek [GR89] is motivated by the observation that it is generally easier to predict serialization order between global transactions that directly conflicts with each other. The approach introduces a synchronization object called ticket in each site, and requires each subtransaction to access the ticket on its site. Ticket is implemented as a timestamp. Each access to the ticket consists of the following two operations: reading the current value and incrementing it by one. These two operations must be included in the code of each subtransaction and are synchronized along with other database read/write operations.
(so that access to these two operations is atomic). It is assumed that each LCC maintains the serializability of its local execution. According to the serializability theory, two conflict transactions are ordered in accordance with the order of their conflicting operations. Since the ticket accessing operations are conflicting operations, the serialization order of the subtransactions is the same as the order that their ticket accessing operations are performed. Furthermore, the ticket are monotonously increased when it is accessed. Therefore, the ticket value accessed reflects the order that the ticket accessing operation is executed, and therefore reflects the serialization order of the subtransaction. The serialization order, after being obtained, can be used by the GCC to validate the execution of the global transactions. The ticket accessing operations are embedded in the code of the subtransactions. No explicit serialization order is needed from the LCC, therefore, no modification of the LCC is required.

The algorithm is interesting and unique in that it maintains the global serializability, does not violate local autonomy and provides a high degree of concurrency. The algorithm addresses the problem of how to effectively obtain the serializations orders of global transactions. A minor problem with this algorithm is that it may abort more global transactions than necessary, due to conflicts introduced by tickets. Since every two global transactions directly conflict with each other at each site they both access, it is very likely that there is a cycle in the global serialization graph, unless some restrictions are imposed on their submission.

5 Conclusion

5.1 Summary

A brief summary of seven algorithms discussed in the paper is given in the following table.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Consistency</th>
<th>Autonomy</th>
<th>Concurrency</th>
<th>Xact abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altruistic Locking</td>
<td>SR/QSR</td>
<td>Design/✓</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Top-down</td>
<td>SR/QSR</td>
<td>Design/✓</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Site graph</td>
<td>SR</td>
<td>✓</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Access graph</td>
<td>SR/QSR</td>
<td>Design/✓</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Decentralized GCC</td>
<td>SR</td>
<td>Static</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>SuperDatabases</td>
<td>SR</td>
<td>Dynamic</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Optimistic ticket</td>
<td>SR</td>
<td>✓</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

In the table, an $SR$ entry in the third column indicates that the algorithm maintains global serializability, while an $SR/QSR$ entry indicates that the algorithm maintains global serializability in some environments, and maintains quasi serializability in general. A check mark $✓$ in the fourth column means that the algorithm preserves local autonomy, while $Static$ (or $Dynamic$) means that the algorithm may violate static (or dynamic) autonomy. A $Design/✓$ entry means that the algorithm may violate design autonomy if serializability is used as the correctness criterion, but preserves local autonomy if quasi serializability is used as the criterion. The concurrency degree
and transaction abort ratio are numbered in order: 1 for the best and 7 for the worst.

Generally, optimistic approaches are superior to pessimistic approaches in concurrency degree, but inferior to in transaction abort ratio. In most HDDBSs applications, the number of global transactions is small in comparison to that of local transactions. Therefore, low degrees of concurrency among global transactions does not affect the overall concurrency very much. On the other hand, a high global transaction abort ratio is usually unacceptable in most HDDBS applications.

One problem with most pessimistic approaches is that they maintain global serializability only if the LDBSs use specific concurrency control strategies. The problem, however, is not that bad in practice. The reason is that the algorithms work for most practically used strategies (e.g., two-phase locking and timestamp ordering protocols).

The key point in success of optimistic approaches is to detect inconsistency between local executions precisely. Although it is very likely that global transactions are scheduled inconsistently at different sites, they usually do not conflict with each other. If the real conflicts can be detected precisely, the approaches will provide good performance. Unfortunately, this is difficult, due to local autonomy.

5.2 Future Work

Global concurrency control is still an active research area. The following are some research directions that we think are interesting.

Pessimistic and optimistic approaches each has advantages over the other in some aspects. An interesting problem is therefore how to combine them together to develop an algorithm that not only provides a high degree of concurrency but also aborts few global transactions.

As in many other research areas, one deficiency in our discussion of global concurrency control is the lack of performance data. Little work has been done in this important area.

Another problem with the existing concurrency control strategies is that most of them are designed independently of other transaction management issues, e.g., commitment and recovery. Clearly, more attention should be devoted to these issues, as well as the integration with concurrency control strategies.

References


